

(incomplete and not always completely accurate)

Notation:

... several things, often repetition of the items before and after it
[...] optional construct, except for its use with arrays
... | ... | ... alternatives, except for its use for the 'or' operation in Boolean expressions
italics font a description of what should appear in a location

Class and interface:

```
[ package mainPackage.subpackage; ]  
[ import mainPackage.subPackage.ClassName; ... ]  
[ public | protected | private ] [ abstract ] class Name [ <GenericType1, ..., <GenericType2> ]  
[ extends Name2 ] [ implements Name3, ... , Name4 ]  
{  
    constructors, constants, fields, methods, and inner classes in any order  
}  
  
[ public ] interface Name [ <GenericType1, ..., <GenericType2> ] [ extends Name3, ..., Name4 ]  
{  
    public constants, public abstract methods, and inner classes in any order  
}
```

Note that each class/interface is in its own file that has the same name as the class/interface and extension .java

Comments:

*/** multi-line comment used for javadocs */* */* multi-line comment */* *// comment for the rest of the line*

Variable declarations:

```
[ public | protected | private ] int i, j = 3, k;                      // other types: byte, short, long, char  
[ public | protected | private ] float x, y = 4.3f;                      // need the "f" to obtain a float  
literal, otherwise double  
[ public | protected | private ] double d, e = 4.3, f = 5e3;  
[ public | protected | private ] boolean a, b = true, c = false;  
[ public | protected | private ] final double MY_PI = 3.14159265;      // constant  
[ public | protected | private ] String s, t = null, u = "Example" ;  
[ public | protected | private ] MyType f, g = null, h = new MyType(...);
```

Constructor and method:

```
[ public | protected | private ] ClassName (Type name, Type name, ... Type name)  
                                                            // need the parenthesis even if no arguments  
{  
    declarations, statements, and inner classes  
}  
  
[ public | protected | private ] [ abstract ] [ static ] [ void | Type ]  
    methodName (Type name, Type name, ... Type name) [ throws exception1, ...  
    exception2 ]  
{  
    declarations, statements, and inner classes  
}
```

Expressions:

Arithmetic operators: + - * /

Note the division of 2 integers results in an integer value obtained by truncating any decimal digits

% remainder (fractional part of a division)

++ unary operator to increment

-- unary operator to decrement

Logical operators: &, && (and), |, || (inclusive or), ! (not), ^ (exclusive or)

Relational operators: <, <=, >, >=, == (no space between them), != // for objects, usually use equals()

(*NewType*) *expression* // cast the expression to type *NewType*; only permitted in certain situations

might be lost. // Any numeric value can be cast to any numeric type, but accuracy

this // The cast is necessary if accuracy might be lost, eg. long to float.

accessorName (*arg1*, ... *arg2*) // the object within which execution is currently taking place

if no arguments // for a routine invocation, need the parenthesis even

BooleanExpression ? *ExpressionOfType1* : *ExpressionOfType1* conditional expression

Statement:

{ ... }

variable = *expression*;

modifierName (*arg1*, ... *arg2*); // need the parenthesis even if no arguments; valid even for accessors

if (*booleanCondition*)

statement1 // use a block for multiple statements

[else

statement2] // use a block for multiple statements

switch (*intExpression*)

{

Case *constantIntExpression* :

0 or more statements, declarations, or inner classes

[break]

...

default:

0 or more statements, declarations, or inner classes

}

break;

while (*booleanCondition*)

do

statement

statement

// use a block for multiple statements

while (*booleanCondition*);

for (*variablesDeclaration* | *assignments*; *booleanCondition*; *assignments* | *increments* | *decrements*)

// multiple assignments, increments or decrements are separated by

commas

statement // use a block for multiple statements

for (*Type identifier* : *instanceOfIterableCollection*) // do the loop for each item in the container

statement // using *identifier* to access the current item

return *expression* ;

throw *exceptionExpression*;

```

try
    block1
catch ( ThrowableType identifier)
    do
...
[ finally block3 ]

```

Arrays: // Note that arrays are reference types, and hence are descendants of the Object class

```

Type[ ] myArray;                Type[ ][ ] twoDArray;
myArray = new Type[length];      myArray.length // yields the length used to
create the array
// Note that the valid index range is 0 to length-1
myArray[index] = value;          myArray = { value1, value2, ... valueLast };

```

Strings:

```

myString = "some " + "characters";
myString.length() // Number of characters in the string; note parentheses for String length
but not array
myString.equals(yourString)      or myString.compareTo(yourString) // don't use
== or !=

```

Object: some methods of the Object class are toString(), equals(), hashCode(), clone(), getClass()